Appendices

A. Extensions to the Formalism

In this appendix, we consider possible extensions to our formalism. These illuminate interesting issues, and the extensions are compatible with our overall approach to modeling. Some of these extensions are already supported in our implementation at https://github.com/HMEIatJHU/neural-datalog-through-time, and more of them may be supported in future versions.

A.1. Cyclicity

Our embedding definitions in §2.2 and §3.1 assumed that the proof graph was acyclic. However, it is possible in general Datalog programs for a fact to participate in some of its own proofs.

For example, the following classical Datalog program finds the nodes in a directed graph that are reachable from the node `start`:

```
reachable(start).
reachable(V) :- reachable(U), edge(U,V).
```

In neural Datalog, the embedding of each fact of the form `reachable(V)` depends on all paths from `start` to `V`. However, if `V` appears on a cycle in the directed graph defined by the `edge` facts, then there will be infinitely many such paths, and our definition of $[\![\texttt{reachable(V)}]\!]$ would then be circular.

Restricting to acyclic proofs. One could define embeddings and probabilities in a cyclic proof graph by considering only the acyclic proofs of each atom h. This is expensive in the worst case, because it can exponentially increase the number of embeddings and probabilities that need to be computed. Specifically, if S is a (finite) set of atoms, let $[\![h/S]\!]$ denote the embedding constructed from acyclic proofs of h that do not use any of the atoms in the finite set S. We define $[\![h/S]\!]$ to be null if $h \in S$, and otherwise to be defined similarly to $[\![h]\!]$ but where equations (4) and (8) are modified to replace each $[\![g_i]\!]$ with $[\![g_i/(S \cup \{h\})]\!]$.[14] As usual, these formulas skip pooling over instantiations where any $[\![\cdot]\!]$ values in the body are null. The recursive definition terminates because S grows at each recursive step but its size is bounded above (§2.7).

[14] For increased efficiency, one can simplify $S \cup \{h\}$ here to eliminate atoms that can be shown by static analysis or depth-first search not to appear in any proof of g_i. This allows more reuse of previously computed $[\![\cdot]\!]$ terms and can sometimes prevent exponential blowup. In particular, if it can be shown that all proofs of h are acyclic, then $[\![h/S]\!]$ can always be simplified to $[\![h/\emptyset]\!]$ and the computation of $[\![h/\emptyset]\!]$ is isomorphic to the ordinary computation of $[\![h]\!]$; the algorithm then reduces to the ordinary algorithm from the main paper.

In particular, this scheme defines $[\![h/\emptyset]\!]$, the **acyclic embedding** of h, which we consider to be an output of the neural Datalog program. Similarly, in neural Datalog through time, the probability of an event e is derived from $\lambda_{e/\emptyset}$, which is computed in the usual way (§3.2) as an extra dimension of the acyclic embedding $[\![e/\emptyset]\!]$.

Forward propagation. This is a more practical approach, used by Hamilton et al. (2017a) to embed the vertices of a graph. This method recomputes all embeddings in parallel, and repeats this for some number of iterations. In our case, for a given time t, each $[\![h]\!]$ is initialized to $\mathbf{0}$, and at each iteration it is recomputed via the formulas of §3.1 and §3.3, using the $[\![g_i]\!]$ values from the previous iteration (also at time t) and the cell block $\boxed{h}$ (determined by events at times $s < t$).

We suggest the following variant that takes the graph structure into account. At time t, construct the (finite) Datalog proof graph, whose nodes are the facts at time t. Visit its strongly connected components in topologically sorted order. Within each strongly connected component C, initialize the embeddings to $\mathbf{0}$ and then recompute them in parallel for $|C|$ iterations. If the graph is acyclic, so that each component C consists of a single vertex, then the algorithm reduces to an efficient and exact implementation of §3.1 and §3.3. In the general case, visiting the components in topologically sorted order means that we wait to work on component C until its strictly upstream nodes have "converged," so that the limited iterations on C make use of the best available embeddings of the upstream nodes. By choosing $|C|$ iterations for component C, we ensure that all nodes in C have a chance to communicate: information has the opportunity to flow end-to-end through all cyclic or acyclic paths of length $< |C|$, and this is enough to include all acyclic paths within C. Note that the embeddings computed by this algorithm (or by the simpler method of Hamilton et al. (2017a)) are well-defined: they depend only on the graph structure, not on any arbitrary ordering of the computations.

A.2. Negation in Conditions

A simple extension to our formalism would allow negation in the body of a rule (i.e., the part of the rule to the right of :- or <-). In rules of the form (1) or (2), each of the conditions condit_i could optionally be preceded by the negation symbol `!`. In general, a rule only applies when the ordinary conditions are true and the negated conditions are false. The concatenation of column vectors in equations (4) and (8) omits $[\![g_i]\!]$ if condit_i is negated, since then g_i is not a fact and does not have a vector (rather, $[\![g_i]\!] = \text{null}$).

Many dialects of Datalog permit programs with negation. If we allow cycles (Appendix A.1), we would impose the usual restriction that negation may not appear on cycles,

i.e., programs may use only **stratified negation**. This restriction ensures that the set of facts is well-defined, by excluding rules like `paradox :- !paradox.`

Example. Extending our example of §2, we might say that a person can eventually grow up into an adult and acquire a gender. Whether person `X` grows up into (say) a woman, and the time at which this happens, depends on the probability or intensity (§3.2) of the `growup(X,female)` event. We use negation to say that a `growup` event can happen only once to a person—after that, all `growup` events for that person become false atoms (have probability 0).

```
24 adult(X,G) <- growup(X,G).
25 adult(X) :- adult(X,G).
26 growup(X,G) :- person(X), gender(G), !adult(X).
27 gender(female).
28 gender(male).
29 gender(nonbinary).
       ⋮
```

As a result, an adult has exactly one gender, chosen stochastically. Female and male adults who know each other can procreate:

```
30 procreate(X,Y) :- rel(X,Y),
       adult(X,female), adult(Y,male).
```

A.3. Highway Connections

As convenient "syntactic sugar," we introduce a variant `:=` of the `:-` connector. The extra horizontal line introduces extra **highway connections** that skip a level in the neural network. A fact's embedding can now be directly affected by its grandparents in the proof DAG, not just its parents. This does not change the set of facts that are proved.

Highway connections of roughly this sort have been argued to help neural network training by providing shorter, more direct paths for backpropagation (Srivastava et al., 2015). They also increase the number of parameters in the model.

We use an example to show how they are specified in neural Datalog. Consider the following `:=` rules. The first rule replaces rule 4 from §2 with a `:=` version. The second rule is added to make the example more interesting. It uses a high-dimensional `teacher` embedding that represents the academic relationship between `X` and `Y` (which is presumably updated by every academic interaction between them).

```
31 rel(X,Y) := opinion(X,U), opinion(Y,U).
32 rel(X,Y) := teacher(X,Y).
```

The embeddings of `rel` facts are computed as before. However, the `:=` rules in the definition of `rel` affect the interpretation of the other `:-`, `:=`, and `<-` rules in the program whose body contains `rel`. A simple example of such a rule is rule 14 from §2.5:

```
33 help(X,Y) :- rel(X,Y).
```

The following rules are now *automatically added to the program*:

```
34 help(X,Y) :- opinion(X,U), opinion(Y,U).
35 help(X,Y) :- teacher(X,Y).
```

As a result, an embedding such as ⟦`help(eve,adam)`⟧ is defined using *not only* ⟦`rel(eve,adam)`⟧, but *also* the embeddings of any lower-level facts that proved `rel(eve,adam)` via the `:=` rules 31 and 32.

In the simple case where `rel(eve,adam)` has only one proof, this scheme is equivalent to augmenting ⟦`rel(eve,adam)`⟧ by concatenating it with the embeddings of its parent or parents. This higher-dimensional version of ⟦`rel(eve,adam)`⟧ now participates as usual in the computation of other embeddings such as ⟦`help(eve,adam)`⟧. However, notice that the dimensionality of the augmented ⟦`rel(eve,adam)`⟧ will differ according to whether `rel(eve,adam)` was proved via rule 31 or rule 32. Therefore, different parameters must be used for the additional dimensions, associated with rule 34 or rule 35 respectively.

More generally, notice that ⟦`help(eve,adam)`⟧ will *sum* over the contributions from the two rules 34 and 35 (via equation (3) or equation (7)). The former contribution may itself involve *pooling* (via equation (4)) over all topics `U` about which `eve` and `adam` both have opinions. This pooling is performed separately from the pooling over `U` used in rule 31: in particular, it may use a different β parameter.

Of course, the definition of `rel` may also include *non-highway* rules such as

```
36 rel(X,Y) :- married(X,U).
37 rel(X,Y) <- hire(X,Y).
```

Since rule 33 is still in the program, however, proving `rel(eve,adam)` remains sufficient to prove the possible event `help(eve,adam)` even when `rel(eve,adam)` is proved by non-highway rules.

Longer highways can be created by chaining multiple `:=` rules together. For example, if we replace rule 33 with a `:=` version,

```
38 help(X,Y) := rel(X,Y).
```

then rules 34–35 will also use `:=`. Hence, any rule whose body uses `help` will automatically acquire versions that mention `rel`, `opinion`, and `teacher` (by repeating the bodies of rules 33–35 respectively).

There are several subtleties in our highway program transformation:

The additional rules 34–35 were constructed by expanding ("inlining") the call to `rel` within the body of rule 33. In logic programming, the inlining transformation is known as **unfolding**. In general it may involve unification, as well as variable renaming to avoid capture.

When we unfold a rule condition, the original condition is usually deleted from the new (unfolded) version of the rule,

since it is now redundant. However, the event that triggers an update rule cannot be deleted in this way. Consider the update rule 11 from §2.3:

```
grateful(Y,X) <- help(X,Y), person(Y).
```

Suppose `help(X,Y)` is defined using the highway rule 38. The rule that we automatically add cannot be

```
grateful(Y,X) <- rel(X,Y), person(Y).
```

as one might expect, because `rel(X,Y)` is not even an event that can be used in this position. Instead, we must ensure that the event is still triggered by the original event:

```
grateful(Y,X) <- help(X,Y) : 0, rel(X,Y),
    person(Y) : 0.
```

As explained in Appendix B below, the `: 0` notation says that although the highway rule 41 is *triggered* by the event `help(X,Y)`, it ignores the event's *embedding*. After all, the event's embedding is still considered by the original rule 39 and does not need to be considered again. The contributions of these two rules will be summed by equation (3) or equation (7) before tanh is applied.

The above example also illustrates the handling of rule conditions that are *not* unfolded, such as `person`. The unfolded rule (e.g., rule 41) marks these conditions with `: 0` as well, to say that while they are still boolean conditions on the update, their embeddings should also be ignored. Again, their embeddings are considered in the original rule 39, so they do not need to be considered again.

Finally, notice that a rule body may contain multiple events and/or conditions that are defined using highway rules. How do we expand

```
world <- e, f, g.
```

given the following highway definitions?

```
e := e1.
e := e2.
g := g1.
g := g2.
```

The general answer is that we unfold each of the body elements in parallel, to allow highway connections from that element. In this case we add 4 new rules:

```
world <- e : 0, e1, f : 0, g : 0.
world <- e : 0, e2, f : 0, g : 0.
world <- e : 0,     f : 0, g1.
world <- e : 0,     f : 0, g2.
```

A.4. Infinite Domains

§2.7 explained that under our current formalism, any given model only allows a finite set of atoms. Thus, it is not possible for new persons to be born.

One way to accommodate that might be to relax Datalog's restriction on nesting.[15] This allows us to build up an infinite set of atoms from a finite set of initial entities:

```
birth(X,Y,child(X,Y)) <- procreate(X,Y).
```

Thus, each new person would be named by a tree giving their ancestry, e.g., `child(eve,adam)` or `child(awan,child(eve,adam))`. But while this method may be useful in other settings, it unfortunately does not allow `eve` and `adam` to have *multiple* children.

Instead, we suggest a different extension, which allows events to create new *anonymous* entities (rather than nested terms):

```
birth(X,Y,*) <- procreate(X,Y).
```

The special symbol `*` denotes a new entity that is created during the update, in this case representing the child being born. Thus, the event `procreate(eve,adam)` will launch the fact `birth(eve,adam,cain)`, where `cain` is some internal name that the system assigns to the new entity. In the usual way when launching a fact, the cell block `birth(eve,adam,cain)` is updated from an initial value of $\mathbf{0}$ by equation (10) in a way that depends on $[\![$`procreate(eve,adam)`$]\!]$.

From the new fact `birth(eve,adam,cain)`, additional rules derive further facts, stating that `cain` is a person and has two parents:[16]

```
person(Z) :- birth(X,Y,Z).
parent(X,Z) :- birth(X,Y,Z).
parent(Y,Z) :- birth(X,Y,Z).
```

Notice that the embedding $[\![$`person(cain)`$]\!]$ initially depends on the state of his parents and their relationship at the time of his procreation. This is because it depends on $[\![$`birth(eve,adam,cain)`$]\!]$ which depends through its cell block on $[\![$`procreate(eve,adam)`$]\!]$, as noted above. $[\![$`person(cain)`$]\!]$ may be subsequently updated over time by events such as `help(eve,cain)`, which affect its cell block.

As another example, here is a description of a sequence of orders in a restaurant:

```
:- embed(dish, 5).
:- event(order, 0).
```

[15] To be safe, we should allow *only* the <- rules (which are novel in our formalism) to derive new facts with greater nesting depth than the facts that appear in the body of the rule. This means that the nesting depth of the database may increase over time, by a finite amount each time an event happens. If we allowed that in traditional :- rules, for example `peano(s(X)) :- peano(X)`, then we could get an infinite set of facts at an *single* time. But then computation at that time might not terminate, and our $\bigoplus^{\beta}$ operators might have to aggregate over infinite sets (see §2.7).

[16] Somewhat awkwardly, under our design, rule 23 is not enough to remove `person(cain)` from the database, since that fact was established by a :- rule. We actually have to write a rule canceling `cain`'s birth: `!birth(X,Y,Z) <- die(Z).` Notice that this rule will remove not only `person(cain)` but also `parent(eve,cain)` and `parent(adam,cain)`. Even then, the entity `cain` may still be referenced in the database as a `parent` of his own children, until they die as well.

```
order(X) :- dish(X).
order(*).
dish(X) <- order(X).
```

This program says that the possible orders consist of any existing dish or a new dish. When used in the discrete-time setting, this model is similar to the Chinese restaurant process (CRP) (Aldous et al., 1985). Just as in the CRP,

- The relative probability of ordering a new dish at time $s \in \mathbb{N}$ is a (learned) constant (because rule 4 has no conditions).
- The relative probability of each possible order(X) event, where X is an existing dish, depends on the embedding of dish(X) (rule 3). That embedding reflects only the number of times X has been ordered previously (rule 5), though its (learned) dependence on that number does not have to be linear as in the CRP.

Interestingly, in the continous-time case—or if we added a rule dish(X) <- tick that causes an update at every discrete time step (see Appendix A.5 below)—the relative probability of the order(X) event would also be affected by the time intervals between previous orders of X. It is also easy to modify this program to get variant processes in which the relative probability of X is also affected by previous orders of dishes Y $\neq$ X (cf. Blei & Lafferty, 2006) or by the exogenous events at the present time and at times when X was ordered previously (cf. Blei & Frazier, 2010).

Appendix A.6 below discusses how an event may trigger an unbounded number of dependent events that provide details about it. This could be used in conjunction with the * feature to create a whole tree of facts that describe a new anonymous entity.

A.5. Uses of Exogenous Events

The extension to allow exogeneous events was already discussed in the main paper (§2.4). Here we mention two specific uses in the discrete-time case.

It is useful in the discrete-time case to provide an exogenous tick event at *every* $s \in \mathbb{N}$. (Note that this results in a second event at every time step; see footnote 11.) Any cell blocks that are updated by the exogenous tick events will be updated even at time steps s between the modeled events that affect those cell blocks. For example, one can write a rule such as person(X) <- tick, person(X), world. so that persons continue to evolve even when nothing is happening to them. This is similar to the way that in the continous-time case, cell blocks with $\delta \neq 0$ will drift via equation (9) during the intervals between the modeled events that affect those cell blocks.[17]

[17] In fact, tick events can *also* be used in the continuous case, if desired (Mei & Eisner, 2017). Then the drifting cells not only drift, but also undergo periodic learned updates that may depend on other facts (as specified by the tick update rules).

Another good use of exogenous events in discrete time is to build a conditional probability model such as a word sequence tagger. At every time step s, a word occurs as an exogenous event, at the same time that the model generates an tag event that supplies a tag for the word at the previous time step. These two events at time s *together* update the state of the model to determine the distribution over the next tag at time $t = s + 1$. Notice that the influences of the word and the tag on the update vector are summed (by the $\sum_r$ in equation (9)). This architecture is similar to a left-to-right LSTM tagger (cf. Ling et al., 2015; Tran et al., 2016).

A.6. Modeling Multiple Simultaneous Events

§3.2 explained how to model a discrete-time event sequence:

> To model a discrete-time event sequence, define the **probability** of an event of type h at time step t to be proportional to $\lambda_e(t)$, normalizing over all event types that are possible then.

In such a sequence, *exactly* one event is generated at each time t. To change this to "*at most* one event," an additional event type none can be used to encode "nothing occurred."

Our continuous-time models are also appropriate for data in which *at most* one event occurs at each time t, since almost surely, there are no times t with multiple events. Recall from §3.2 that in this setting, the expected number of occurrences of e on the interval $[t, t + dt)$, divided by dt, approaches $\lambda_e(t)$ as $dt \to 0^+$. Thus, given a time t at which one event occurs, the expected total number of *other* events on $[t, t + dt)$ approaches 0 as $dt \to 0^+$.

However, there exist datasets in which multiple events do occur at time t—even multiple copies of the same event. By extending our formalism with a notion of **dependent events**, we can model such datasets generatively. The idea is that an event e at time t can stochastically generate dependent events that also occur at time t.

(When multiple events occur at time t, our model already specifies how to handle the <- rule updates that result from these events. Specifically, multiple events that simultaneously update the same head are pooled within and across rules by equation (9).)

To model the events that depend on e, we introduce the notion of an **event group**, which represents a group of competing events at a particular instant. Groups do not persist over time; they appear momentarily in response to particular events. If event e at time t **triggers** group g and g is

non-empty at time t, then exactly one event e' in g (perhaps none) will stochastically occur at time t as well.

Under some programs, it will be possible for multiple copies—that is, **tokens**—of the *same* event type to occur at the same time. For precision, we use e below for a particular event token at a particular time, using $\bar{e}$ to denote the Datalog atom that names its event type. Similarly, we use g for a particular token of a triggered group, using $\bar{g}$ to denote the Datalog atom that names the type of group. We write $[\![e]\!]$ and $[\![g]\!]$ for the token embeddings: this allows different tokens of the same type to have different embeddings at time t, depending on how they arose.

We allow new program lines of the following forms:[18]

```
:- eventgroup(functor, dimension).          (13a)
group <<- event, condit_1, ..., condit_N.   (13b)
event <-- group, condit_1, ..., condit_N.   (13c)
```

An `eventgroup` declaration of the form (13a) is used to declare that atoms with a particular functor refer to event groups, similar to an `event` declaration. We will display such functors with a double underline.

A rule of the form (13b) is used to trigger a group of possible dependent events. If e is an event token at time t, then it triggers a token g of group type $\bar{g}$ at time t, for each $\bar{g}$ and each rule r having at least one instantiation of the form $\bar{g}$ <<- $\bar{e}$, c_1, ..., c_N for which the c_i are all facts at time t. The embedding of this group token g pools over all such instantiations of rule r (as in equation (4)):

$$[\![g]\!] \stackrel{\text{def}}{=} \bigoplus_{c_1,\ldots,c_N}^{\beta_r} \mathbf{W}_r \underbrace{[1; [\![e]\!]; [\![c_1]\!]; \ldots; [\![c_N]\!]]}_{\text{concatenation of column vectors}} \in \mathbb{R}^{D_g} \quad (14)$$

where all embeddings are evaluated at time t.

Rules of the form (13c) are used to specify the possible events in a group. Very similarly to the above, if the group g is triggered at time t, then it contains a token e' of event type $\bar{e}'$, for each $\bar{e}'$ and each rule r having at least one instantiation of the form $\bar{e}'$ <-- $\bar{g}$, c_1, ..., c_N for which the c_i are all facts at time t. The embedding of this event token e' pools over all such instantiations of rule r:

$$[\![e']\!] \stackrel{\text{def}}{=} \bigoplus_{c_1,\ldots,c_N}^{\beta_r} \mathbf{W}_r \underbrace{[1; [\![g]\!]; [\![c_1]\!]; \ldots; [\![c_N]\!]]}_{\text{concatenation of column vectors}} \in \mathbb{R}^{D_g} \quad (15)$$

where all embeddings are evaluated at time t.

Since each e' in group g is an event, we compute not only an embedding $[\![e']\!]$ but also an unnormalized probability $\lambda_{e'}$, computed just as in §3.2 (using exp rather than softplus). Exactly one of the finitely many event tokens in g will occur at time t, with event type e' being chosen from g with probability proportional to $\lambda_{e'}$.

Training. In fully supervised training of this model, the dependencies are fully observed. For each dependent event token e' that occurs at time t, the training set specifies what it depends on—that it is a dependent event, which group g it was chosen from, and which rule r established that e' was an element of g. Furthermore, the training set must specify for g which event e triggered it and via which rule r. However, if these dependencies are not fully observed, then it is still possible to take the training objective to be the incomplete-data likelihood, which involves computing the total probability of the bag of events at each time t by summing over all possible choices of the dependencies.

Marked events. To see the applicability of our formalism, consider a marked point process (such as the marked Hawkes process). This is a traditional type of event sequence model in which each event occurrence also generates a stochastic **mark** from some distribution. The mark contains details about the event. For example, each occurrence of `eat_meal(eve)` might generate a mark that specifies the food eaten and the location of the meal.

Why are marked point processes used in practice? An alternative would be to refine the atoms that describe events so that they contain the additional details. This leads to fine-grained event types such as `eat_meal(eve,apple,tree_of_knowledge)`. However, that approach means that computing $\lambda(t) \stackrel{\text{def}}{=} \sum_{e \in \mathcal{E}(t)} \lambda_e(t)$ during training (§4) or sampling (Appendix F.2) involves summing over a large set of fine-grained events, which is computationally expensive. Using marks makes it possible to generate a coarse-grained event first, modeling its probability without yet considering the different ways to refine it. The event's details are considered only once the event has been chosen. This is simply the usual computational efficiency argument for locally normalized generative models.

Our formalism can treat an event's mark as a dependent event, using the neural architecture above to model the mark probability $p(e' \mid e)$ as proportional to $\lambda_{e'}$. The set of possible marks for an event is defined by rules of the form (13) and may vary by event type and vary by time.

Multiply marked events. Our approach also makes it easy for an event to independently generate multiple marks, which describe different attributes of an event. For example, each meal at time t may select a dependent location,

```
1 :- eventgroup(restaurants, 5).
2 :- event(eat_at, 0).
3 restaurants <<- eat_meal(X).
4 eat_at(Y) <-- restaurants, is_restaurant(Y).
5 eat_at(home) <-- restaurants.
```

which associates some dependent restaurant `Y` (or `home`)

[18] Mnemonically, note that the "doubled" side of the symbol <<- or <-- is next to the group, since the group usually contains multiple events. This is also why group names are double-underlined in the examples below.

with the meal.[19] At the same time, the meal may select a *set* of foods to eat, where each food `U`[20] is in competition with `none`[21] to indicate that it may or may not be chosen:

```
:- eventgroup(optdish, 7).
:- event(eat_dish, 0).
:- event(none, 0).
optdish(U) <<- eat_meal(X),
      food(U), opinion(X,U).
eat_dish(U) <-- optdish(U).
none <-- optdish(U) : 0.
```

Recursive marks. Dependent events can recursively trigger dependent events of their own, leading to a tree of event tokens at time t. This makes it possible to model the top-down generation of tree-structured metadata, such as a syntactically well-formed sentence that describes the event (Zhang et al., 2016). Observing such sentences in training data would then provide evidence of the underlying embeddings of the events. For example, to generate derivation trees from a context-free grammar, encode each nonterminal symbol as an event group, whose events are the production rules that can expand that nonterminal. In general, the probability of a production rule depends on the sequence of production rules at its ancestors, as determined by a recurrent neural net.

A special case of a tree is a sequence: in the meal example, each dish could be made to generate the next dish until the sequence terminates by generating `none`. The resulting architecture precisely mimics the architecture of an RNN language model (Mikolov et al., 2010).

Multiple agents. A final application of our model is in a discrete-time setting where there are multiple agents, which naturally leads to multiple simultaneous events. For example, at each time step t, every person stochastically chooses an action to perform (possibly `none`). This can be accomplished by allowing the `tick` event (Appendix A.5) to trigger one group for each person:

```
:- eventgroup(actions, 7).
actions(X) <<- tick, person(X).
help(X,Y) <-- actions(X), rel(X,Y).
      ⋮
```

This is a group-wise version of rule 14 in the main paper.

A similar structure can be used to produce a "node classification" model in which each node in a graph stochastically generates a label at each time step, based on the node's current embedding (Hamilton et al., 2017b; Xu et al., 2020). The event group for a node contains its possible labels. The graph structure may change over time thanks to exogeneous or endogenous events.

Example. For concreteness, below is a fully generative model of a dynamic colored directed graph, using several of the extensions described in this appendix. The model can be used in either a discrete-time or continuous-time setting.

The graph's nodes and edges have embeddings, as do the legal colors for nodes:

```
:- embed(node, 8).
:- embed(edge, 4).
:- embed(color, 3).
```

In this version, edges are stochastically added and removed over time, one at a time. Any two unconnected nodes determine through their embeddings the probability of adding an edge between them, as well as the initial embedding of this edge. The edge's embedding may drift over time,[22] and at any time determines the edge's probability of deletion.

```
:- event(add_edge, 8).
:- event(del_edge, 0).
add_edge(U,V) :- node(U), node(V), !edge(U,V).
del_edge(U,V) :- edge(U,V).
edge(U,V) <- add_edge(U,V).
!edge(U,V) <- del_edge(U,V).
```

Adding `edge(U,V)` to the graph causes two dependent events that simultaneously and stochastically relabel both `U` and `V` with new colors. This requires triggering two event groups (unless `U=V`). A node's new color `C` depends stochastically on the embeddings of the node and its neighbors, as well as the embeddings of the colors:

```
:- eventgroup(labels, 8).
:- event(label, 8).
labels(U) <<- add_edge(U,V).
labels(V) <<- add_edge(U,V).
label(X,C) <-- labels(X), color(C), node(X),
      edge(X,Y), node(Y).
```

Finally, here is how a relabeling event does its work. The `has_color` atoms that are updated here are simply facts that record the current coloring, with no embedding. However, the rules below ensure that a *node*'s embedding records its *history* of colors (and that it has only one color at a time):

```
!has_color(U,D) <- label(U,C), color(D).
has_color(U,C) <- label(U,C).
node(U) <- has_color(U,C), color(C).
```

The initial graph at time $t = 0$ can be written down by enumeration:

[19] Notice that the choice of event `eat_at(Y)` depends on the person `X` who is eating the meal, through the embedding of this token of ⟦`restaurants`⟧, which depends on ⟦`eat_meal(X)`⟧.

[20] Notice that the unnormalized probability of including `U` in `X`'s meal depends on `X`'s opinion of `U`.

[21] The annotation `: 0` in the last line (explained in Appendix B below) is included as a matter of good practice. In keeping with the usual practice in binary logistic regression, it simplifies the computation of the normalized probabilities, without loss of generality, by ensuring that the unnormalized probability of `none` is constant rather than depending on `U`.

[22] In the continuous-time setting, the drift is learned. In the discrete-time setting, we must explicitly specify drift as explained in Appendix A.5, via a rule such as `edge(U,V) <- tick`.

```
18 color(red).
19 color(green).
20 color(blue).
21 has_color(0,red).
22 has_color(1,blue)
23 has_color(2,red).
24 node(U) :- has_color(U,C).
25 edge(0,1) <- init.
```

Inheritance. As a convenience, we allow an event group to be used anywhere that an event can be used—at the start of the body of a rule of type (2a), (2b), or (13b). Such a rule applies at times when the group is triggered (just as a rule that mentions an event, instead of a group, would apply at times when that event occurred).

This provides a kind of inheritance mechanism for events:

```
47 :- eventgroup(act, 5).
48 act(X) <<- sleep(X).
49 act(X) <<- help(X,Y), person(Y).
       ⋮
50 person(Y) <- act(X), parent(X,Y), person(Y).
51 animal(Y) <- act(X), own(X,Y), animal(Y).
```

This means that whenever X takes any action—sleep, help, etc.—rules 50–51 will update the embeddings of X's children and pets.

Adopting the terminology of object-oriented programming, act(eve) functions as a class of events (i.e., event type), whose subclasses include help(eve,adam) and many others. In this view, each particular instance (i.e., event token) of the subclass help(eve,adam) has a method that returns its embedding in $\mathbb{R}^{D_{\texttt{help}}}$. But rules 50–51 instead view this help(eve,adam) event as an instance of the superclass act(eve), and hence call a method of that superclass to obtain the embedding of the group token act(eve) in $\mathbb{R}^{D_{\texttt{act}}} = \mathbb{R}^5$, as defined via equation (14).

In the above example, the event group is actually empty, as there are no rules of type (13c) that populate it with dependent events. Thus, no dependent events occur as a result of the group being triggered. The empty event group is simply used as a class. One could, however, add rules such as

```
52 act_at(L) <-- act(X), location(L).
```

which marks each action (of any type) with a location.

B. Parameter Sharing Details

Throughout §3, the parameters $\mathbf{W}$ and β are indexed by the rule number r. (They appear in equations (4) and (8).) Thus, the number of parameters grows with the number of rules in our formalism. However, we also allow further flexibility to *name* these parameters with atoms, so that they can be shared among and within rules.

This is achieved by explicitly naming the parameters to be used by a rule:

```
head : beta :-
        : bias_vector
        condit_1 : matrix_1,
            ⋮
        condit_N : matrix_N.
```

Now β_r in equation (4) is replaced by a scalar parameter named by the atom *beta*. Similarly, the affine transformation matrix $\mathbf{W}_r$ in equation (4) is replaced by a parameter matrix that is constructed by *horizontally* concatenating the column vector and matrices named by the atoms *bias_vector*, $\textit{matrix}_1, \ldots, \textit{matrix}_N$ respectively.

To be precise, $\textit{matrix}_i$ will have $D_{\textit{head}}$ rows and $D_{\textit{condit}_i}$ columns. The computation (4) can be viewed as multiplying this matrix by the vector embedding of the atom that instiatiates $\textit{condit}_i$, yielding a vector in $\mathbb{R}^{D_{\textit{head}}}$. It then sums these vectors for $i = 1, \ldots, N$ as well as the bias vector (also in $\mathbb{R}^{D_{\textit{head}}}$), obtaining a vector in $\mathbb{R}^{D_{\textit{head}}}$ that it provides to the pooling operator.

These parameter annotations with the : symbol are optional (and were not used in the main paper). If any of them is not specified, it is set automatically to be rule- and position-specific: in the r^{th} rule, *beta* defaults to params(r,beta), *bias_vector* defaults to params(r,bias), and $\textit{matrix}_i$ defaults to params(r,i).

As shorthand, we also allow the form

```
head : beta :-
        condit_1, condit_N :: full_matrix.
```

where *full_matrix* directly names the concatenation of matrices that replaces $\mathbf{W}_r$.

The parameter-naming mechanism lets us share parameters across rules by reusing their names. For example, blessings and curses might be inherited using the same parameters:

```
53 cursed(Y) :- cursed(X),parent(X,Y) :: inherit.
54 blessed(Y) :- blessed(X),parent(X,Y) :: inherit.
```

Conversely, to do *less* sharing of parameters, the parameter names may mention variables that appear in the head or body of the rule. In this case, different instantiations of the rule may invoke different parameters. (*beta* is only allowed to contain variables that appear in the head, because each way of instantiating the head needs a single β to aggregate over all the compatible instantations of its body.)

For example, we can modify rules 53 and 54 into

```
55 cursed(Y) : descendant(Y) :-
       cursed(X), parent(X,Y) :: inherit(X,Y).
56 blessed(Y) : descendant(Y) :-
       blessed(X), parent(X,Y) :: inherit(X,Y).
```

Now each X, Y pair has its own $\mathbf{W}$ matrix (shared by curses and blessings), and similarly, each Y has its own β scalar. This example has too many parameters to be practical, but serves to illustrate the point.

If X or Y is an entity created by the * mechanism (Appendix A.4), then the name will be constructed using a literal *, so that all newly created entities use the same parameters. This ensures that the number of parameters is finite even if the number of entities is unbounded. As a result, parameters can be trained by maximum likelihood and reused every time a sequence is sampled, even though different sequences may have different numbers of entities. Although novel entities share parameters, facts that differ only in their novel entities may nonetheless come to have different embeddings if they are created or updated in different circumstances.

The special parameter name 0 says to use a zero matrix:

```
57 cursed(Y) : descendant :-
       : inherit_bias,
       cursed(X) : inherit,
       parent(X,Y) : 0.
```

In this example, the condition `parent(X,Y)` must still be non-null for the rule to apply, but we ignore its embedding.

The same mechanism can be used to name the parameters of $\leftarrow$ rules. In this case, *event* at the start of the body can also be annotated, as *event* : $matrix_0$. The horizontal concatenation of named matrices now includes the matrix named by $matrix_0$, and is used to replace $\mathbf{W}_r$ in equation (8).

For a $\leftarrow$ rule, it might sometimes be desirable to allow finer-grained control over how the rule affects the drift of a cell block over time (see equation (17) in Appendix C below). For example, forcing $\underline{\mathbf{f}} = \mathbf{1}$ and $\underline{\mathbf{i}} = \mathbf{0}$ in equation (18) ensures via equation (19) that when the rule updates h, it will not introduce a discontinuity in the $\boxed{h}(t)$ function, although it might change the function's asymptotic value and decay rate. (This might be useful for the tick rules mentioned in footnote 17, for example.) Similarly, forcing $\bar{\mathbf{f}} = \mathbf{1}$ and $\bar{\mathbf{i}} = \mathbf{0}$ in equation (18) ensures via equation (20) that the rule does not change the asymptotic value of the $\boxed{h}(t)$ function. These effects can be accomplished by declaring that certain values are $\pm\infty$ in the first column of $\mathbf{W}_r$ in equation (8) (as this column holds bias terms). We have not yet designed a syntax for such declarations.

We can also name the softplus temporal scale parameter τ in §3.2. For example, we can rewrite rule 13 of §2.4 as

```
58 :- event(help, 8) : intervene.
```

and allow harm to share τ with help:

```
59 :- event(harm, 8) : intervene.
```

C. Updating Drift Functions in the Continuous-Time LSTM

Here we give the details regarding continuous-time LSTMs, which were omitted from §3.3 due to space limitations. We follow the design of Mei & Eisner (2017), in which each cell changes endogenously between updates, or "drifts," according to an exponential decay curve:

$$c(t) \stackrel{\text{def}}{=} \bar{c} + (\underline{c} - \bar{c}) \exp(-\delta(t-s)) \quad \text{where } t > s \tag{16}$$

This curve is parameterized by $(s, \underline{c}, \bar{c}, \delta)$, where

- s is a starting time—specifically, the time when the parameters were last updated
- $\underline{c}$ is the starting cell value, i.e., $c(s) = \underline{c}$
- $\bar{c}$ is the asymptotic cell value, i.e., $\lim_{t\to\infty} c(t) = \bar{c}$
- $\delta > 0$ is the rate of decay toward the asymptote; notice that the derivative $c'(t) = \delta \cdot (\bar{c} - \underline{c})$

In the present paper, we similarly need to define the trajectory through $\mathbb{R}^{D_h}$ of the cell block $\boxed{h}$ associated with fact h. That is, we need to be able to compute $\boxed{h}(t) \in \mathbb{R}^{D_h}$ for any t. Since $\boxed{h}$ is not a single cell but rather a block of D_h cells, it actually needs to store not 4 parameters as above, but rather $1+3D_h$ parameters. Specifically, it stores $s \in \mathbb{R}$, which is the time that the block's parameters were last updated: this is shared by all cells in the block. It also stores vectors that we refer to as $\boxed{h}^{\underline{\mathbf{c}}}, \boxed{h}^{\bar{\mathbf{c}}}, \boxed{h}^{\boldsymbol{\delta}} \in \mathbb{R}^{D_h}$. Now analogously to equation (16), we define the trajectory of the cell block elementwise:

$$\boxed{h}(t) \stackrel{\text{def}}{=} \boxed{h}^{\bar{\mathbf{c}}} + (\boxed{h}^{\underline{\mathbf{c}}} - \boxed{h}^{\bar{\mathbf{c}}}) \exp(-\boxed{h}^{\boldsymbol{\delta}} \cdot (t-s)), \tag{17}$$

for all $t > s$ (up to and including the time of the next event that results in updating the block's parameters).

We now describe exactly *how* the block's parameters are updated when an event occurs at time s. Recall that for the discrete-time case, for each (r, m), we obtained $[h]^{\leftarrow}_{rm} \in \mathbb{R}^{3D_h}$ by evaluating (8) at time s. We then set $(\mathbf{f}; \mathbf{i}; \mathbf{z}) \stackrel{\text{def}}{=} \sigma([h]^{\leftarrow}_{rm})$. In the continuous-time case, we evaluate (8) at time s to obtain $[h]^{\leftarrow}_{rm} \in \mathbb{R}^{7D_h}$ (so $\mathbf{W}_r$ needs to have more rows), and accordingly obtain 7 vectors in $(0,1)^{D_h}$,

$$(\underline{\mathbf{f}}; \underline{\mathbf{i}}; \underline{\mathbf{z}}; \bar{\mathbf{f}}; \bar{\mathbf{i}}; \bar{\mathbf{z}}; \mathbf{d}) \stackrel{\text{def}}{=} \sigma([h]^{\leftarrow}_{rm}) \tag{18}$$

which we use similarly to equation (11) to define update vectors for the current cell values (time s) and the asymptotic cell values (time ∞), respectively

$$[h]^{\Delta\underline{\mathbf{c}}}_{rm} \stackrel{\text{def}}{=} (\underline{\mathbf{f}} - 1) \cdot \boxed{h}(s) + \underline{\mathbf{i}} \cdot (2\underline{\mathbf{z}} - 1) \tag{19}$$

$$[h]^{\Delta\bar{\mathbf{c}}}_{rm} \stackrel{\text{def}}{=} (\bar{\mathbf{f}} - 1) \cdot \boxed{h}^{\bar{\mathbf{c}}} + \bar{\mathbf{i}} \cdot (2\bar{\mathbf{z}} - 1) \tag{20}$$

as well as a vector of proposed decay rates:[23]

$$[h]^{\boldsymbol{\delta}}_{rm} \stackrel{\text{def}}{=} \text{softplus}_1(\sigma^{-1}(\mathbf{d})) \quad \in \mathbb{R}^{D_h}_{>0} \tag{21}$$

We then pool the update vectors from different (r, m) and apply this pooled update, much as we did for the discrete-time cell values in equations (9)–(11):

$$\boxed{h}^{\underline{\mathbf{c}}} \stackrel{\text{def}}{=} \boxed{h}(s) + \sum_r \bigoplus_m{}^{\beta_r} [h]^{\Delta\underline{\mathbf{c}}}_{rm} \tag{22}$$

$$\boxed{h}^{\bar{\mathbf{c}}} \stackrel{\text{def}}{=} \boxed{h}^{\bar{\mathbf{c}}} + \sum_r \bigoplus_m{}^{\beta_r} [h]^{\Delta\bar{\mathbf{c}}}_{rm} \tag{23}$$

The special cases mentioned just below the update (9) are also followed for the updates (22)–(23).

The final task is to pool the decay rates to obtain $\boxed{h}^{\boldsymbol{\delta}}$. It is less obvious how to do this in a natural way. Our basic idea is that for the i^{th} cell, we should obtain the decay rate $(\boxed{h}^{\boldsymbol{\delta}})_i$ by a weighted harmonic mean of the decay rates $([h]^{\boldsymbol{\delta}}_{rm})_i$ that were proposed by different (r, m) pairs. A given (r, m) pair should get a high weight in this harmonic mean to the extent that it contributed large updates $([h]^{\Delta\underline{\mathbf{c}}}_{rm})_i$ or $([h]^{\Delta\bar{\mathbf{c}}}_{rm})_i$.

Why harmonic mean? Observe that the exponential decay curve (16) has a **half-life** of $\frac{\ln 2}{\delta}$. In other words, at any moment t, it will take time $\frac{\ln 2}{\delta}$ for the curve to travel halfway from its current value $c(t)$ to $\bar{c}$. (This amount of time is independent of t.) Thus, saying that the decay rate is a weighted harmonic mean of proposed decay rates is equivalent to saying that the half-life is a weighted arithmetic mean of proposed half-lives,[24] which seems like a reasonable pooling principle.

Thus, operating in parallel over all cells i by performing the following vector operations elementwise, we choose

$$\boxed{h}^{\boldsymbol{\delta}} \stackrel{\text{def}}{=} \left(\frac{\sum_r \sum_m \mathbf{w}_{rm} \cdot ([h]^{\boldsymbol{\delta}}_{rm})^{-1}}{\sum_r \sum_m \mathbf{w}_{rm}} \right)^{-1} \tag{24}$$

We define the vector of unnormalized non-negative weights $\mathbf{w}_{rm}$ from the updated $\boxed{h}^{\underline{\mathbf{c}}}$ and $\boxed{h}^{\bar{\mathbf{c}}}$ values by

$$\mathbf{w}_{rm} \stackrel{\text{def}}{=} \left(\bigoplus_{m'}{}^{\beta_r} [h]^{\Delta\underline{\mathbf{c}}}_{rm'} \right) \cdot \frac{[h]^{\Delta\underline{\mathbf{c}}}_{rm}{}^{\beta_r}}{\sum_{m'} [h]^{\Delta\underline{\mathbf{c}}}_{rm'}{}^{\beta_r}}$$

$$+ \left(\bigoplus_{m'}{}^{\beta_r} [h]^{\Delta\bar{\mathbf{c}}}_{rm'} \right) \cdot \frac{[h]^{\Delta\bar{\mathbf{c}}}_{rm}{}^{\beta_r}}{\sum_{m'} [h]^{\Delta\bar{\mathbf{c}}}_{rm'}{}^{\beta_r}}$$

$$+ \boxed{h}^{\bar{\mathbf{c}}} - \boxed{h}^{\underline{\mathbf{c}}} \tag{25}$$

The following remarks should be read elementwise, i.e., consider a particular cell i, and read each vector $\mathbf{x}$ as referring to the scalar $(\mathbf{x})_i$.

The weights defined in equation (25) are valid weights to use for the weighted harmonic mean (24):

- $\mathbf{w}_{rm} \geq 0$, because of the use of absolute value.
- $\mathbf{w}_{rm} > 0$ strictly unless $\boxed{h}^{\bar{\mathbf{c}}} = \boxed{h}^{\underline{\mathbf{c}}}$. Thus, the decay rate $\boxed{h}^{\boldsymbol{\delta}}$ as defined by equation (24) can only be undefined (that is, $\frac{0}{0}$) if $\boxed{h}^{\bar{\mathbf{c}}} = \boxed{h}^{\underline{\mathbf{c}}}$, in which case that decay rate is irrelevant anyway.

The way to understand the first line of equation (25) is as a heuristic assessment of how much the cell's curve (16) was affected by (r, m) via $[h]^{\Delta\underline{\mathbf{c}}}_{rm}$'s effect on $\boxed{h}^{\underline{\mathbf{c}}}$. First of all, $\left(\bigoplus_{m'}{}^{\beta_r} [h]^{\Delta\underline{\mathbf{c}}}_{rm'} \right)$ is the pooled magnitude of *all* of the r^{th} rule's attempts to affect $\boxed{h}^{\underline{\mathbf{c}}}$. Using the absolute value ensures that even if large-magnitude attempts of opposing sign canceled each other out in equation (22), they are still counted here as large attempts, and thus give the r^{th} rule a stronger total voice in determining the decay rate $\boxed{h}^{\boldsymbol{\delta}}$. This pooled magnitude for the r^{th} rule is then partitioned among the attempts (r, m). In particular, the fraction in the first line denotes the portion of the r^{th} rule's pooled effect on $\boxed{h}^{\underline{\mathbf{c}}}$ that should be heuristically attributed to (r, m) specifically, given the way that equation (22) pooled over all m (recall that this invokes equation (6a)).

Thus, the first line of equation (25) considers the effect of (r, m) on $\underline{\mathbf{c}}$. The second line adds its effect on $\bar{\mathbf{c}}$. The third line effectively acts as smoothing so that we do not pay undue attention to the size ratio among different updates if these updates are tiny. In particular, if all of the updates $[h]^{\Delta\underline{\mathbf{c}}}_{rm}$ and $[h]^{\Delta\bar{\mathbf{c}}}_{rm}$ are small compared to the total height of the curve, namely $\boxed{h}^{\bar{\mathbf{c}}} - \boxed{h}^{\underline{\mathbf{c}}}$, then the third line will dominate the definition of the weights $\mathbf{w}_{rm}$, making them close to uniform. The third line is also what prevents inappropriate division by 0 (see the second bullet point above).

D. Likelihood Computation Details

In this section we discuss the log-likelihood formulas in §4.

For the discrete-time setting, the formula simply follows from the fact that the log-probability of event $\boldsymbol{e}$ at time t was defined to be $\log(\lambda_{\boldsymbol{e}}(t)/\lambda(t))$.

The log-likelihood formula (12) for the continuous-time case has been derived and discussed in previous work (Hawkes, 1971; Liniger, 2009; Mei & Eisner, 2017). Intuitively, during parameter training, each $\log \lambda_{\boldsymbol{e}_i}(t_i)$ is

[23] Equation (21) simply replaces the σ that produced $\mathbf{d}$ with softplus_1 (defined in §3.2), since there is no reason to force decay rates into $(0, 1)$.

[24] It is also equivalent to saying that the (⅔)-life is a weighted arithmetic mean of proposed (⅔)-lives, since equation (16) has a (⅔)-life of $\frac{\ln 3}{\delta}$. In other words, there is nothing special about the fraction $1/2$. *Any* choice of fraction would motivate using the harmonic mean.

increased to explain why e_i happened at time t_i while $\int_{t=0}^{T} \lambda(t)\mathrm{d}t$ is decreased to explain why no event of any possible type $e \in \mathcal{E}(t)$ ever happened at other times. Note that there is no log under the integral in equation (12), in contrast to the discrete-time setting.

As discussed in §4, the integral term in equation (12) is computed using the Monte Carlo approximation detailed by Algorithm 1 of Mei & Eisner (2017), which samples times t.

However, at each sampled time t, that method still requires a summation over all events to obtain $\lambda(t)$. This summation can be expensive when there are many event types. Thus, we estimate the sum using a simple downsampling trick, as follows. At any time t that is sampled to compute the integral, let $\mathcal{E}(t)$ be the set of *possible* event types under the database at time t. We construct a bag $\mathcal{E}'(t)$ by uniformly sampling event types from $\mathcal{E}(t)$ with replacement, and estimate

$$\lambda(t) \approx \frac{|\mathcal{E}|}{|\mathcal{E}'|} \sum_{e \in \mathcal{E}'} \lambda_e(t)$$

This estimator is unbiased yet remains much less expensive to compute especially when $|\mathcal{E}'| \ll |\mathcal{E}|$. In our experiments, we took $|\mathcal{E}'| = 10$ and still found empirically that the variance of the log-likelihood estimate (computed by running multiple times) was rather small.

Another computational expense stems from the fact that we have to make Datalog queries after every event to figure out the proof DAG of each provable Datalog atom. Queries can be slow, so rather than repeatedly making a given query, we just memoize the result the first time and look it up when it is needed again (Swift & Warren, 2012). However, as events are allowed to change the database, results of some queries may also change, and thus the memos for those queries become incorrect (stale). To avoid errors, we currently flush the memo table every time the database is changed. This obviously reduces the usefulness of the memos. An implementation improvement for future work is to use more flexible strategies that create memos and update them incrementally through change propagation (Acar & Ley-Wild, 2008; Hammer, 2012; Filardo & Eisner, 2012).

E. How to Predict Events

Figures 2 and 4 include a task-based evaluation where we try to predict the *time* and *type* of the next event. More precisely, for each event in each held-out sequence, we attempt to predict its time given only the preceding events, as well as its type given both its true time and the preceding events.

These figures evaluate the time prediction with average L_2 loss (yielding a root-mean-squared error, or **RMSE**) and evaluate the argument prediction with average 0-1 loss (yielding an **error rate**).

To carry out the predictions, we follow Mei & Eisner (2017) and use the minimum Bayes risk (MBR) principle to predict the time and type with lowest expected loss. To predict the i^{th} event:

- Its time t_i has density $p_i(t) = \lambda(t) \exp(-\int_{t_{i-1}}^{t} \lambda(t')\mathrm{d}t')$. We choose $\int_{t_{i-1}}^{\infty} t p_i(t)\mathrm{d}t$ as the time prediction because it has the lowest expected L_2 loss. The integral can be estimated using i.i.d. samples of t_i drawn from $p_i(t)$ as detailed in Mei & Eisner (2017) and Mei et al. (2019).
- Since we are given the next event time t_i when predicting the type e_i,[25] the most likely type is simply $\arg\max_{e \in \mathcal{E}(t_i)} \lambda_e(t_i)$.

Notice that our approach will never predict an impossible event type. For example, `help(eve,adam)` won't be in $\mathcal{E}(t_i)$ and thus will have *zero* probability if $[\![\texttt{rel(eve,adam)}]\!](t_i) = \textsf{null}$ (maybe because `eve` stops having `opinion`s on anything that `adam` does anymore).

In some circumstances, one might also like to predict the most likely type out of a *restricted* set $\mathcal{E}'(t_i) \subsetneq \mathcal{E}(t_i)$. This allows one to answer questions like "If we know that some event `help(eve,Y)` happened at time t_i, then which person `Y` did `eve` `help`, given all past events?" The answer will simply be $\arg\max_{e \in \mathcal{E}'(t_i)} \lambda_e(t_i)$.

As another extension, Mei et al. (2019) show how to predict missing events in a neural Hawkes process conditioned on partial observations of both past and future events. They used a particle smoothing technique that had previously been used for discrete-time neural sequence models (Lin & Eisner, 2018). This technique could also be extended to neural Datalog through time (NDTT):

- In **particle filtering**, each particle specifies a hypothesized complete history of past events (both observed and missing). In our setting, this provides enough information to determine the set of possible events $\mathcal{E}(t)$ at time t, along with their embeddings and intensities.
- **Neural particle smoothing** is an extension where the guess of the next event is also conditioned on the sequence of future events (observed only), using a learned neural encoding of that sequence. In our setting, it is not clear what embeddings to use for the future events, as we do not in general have static embeddings for our event types, and their dynamic

[25] Mei & Eisner (2017) also give the MBR prediction rule for predicting e_i *without* knowledge of its time t_i.

embeddings cannot yet be computed at time t. We would want to learn a compositional encoding of future events that at least respects their structured descriptions (e.g., `help(eve,adam)`), and possibly also draws on the NDTT program and its parameters in some way. We leave this design to future work.

F. Experimental Details

F.1. Dataset Statistics

Table 1 shows statistics about each dataset that we use in this paper (§6).

F.2. Details of Synthetic Dataset and Models

We synthesized data for §6.1 by sampling event sequences from the structured NHP specified by our Datalog program in that section. We chose $N = 4$ and $M = 4, 8, 16$, and thus end up with three different datasets.

For each M, we set the sequence length $I = 21$ and then used the thinning algorithm (Mei & Eisner, 2017; Mei et al., 2019) to sample the first I events over $[0, \infty)$. We set $T = t_I$, i.e., the time of the last generated event. We generated 2000, 100 and 100 sequences for each training, dev and test set respectively. We showed the learning curves for $M = 8$ and 16 in Figure 1 and left out the plot for $M = 4$ because it is boringly similar.

For the unstructured NHP baseline, the program given in §6.1 is not quite accurate. To exactly match the architecture of Mei & Eisner (2017), we have to use the notation of Appendix B to ensure that each of the MN event types uses its its own parameters for its embedding and probability:

```
1 is_process(1).          3 is_type(1).
      ⋮                         ⋮
2 is_process(M).          4 is_type(N).

5 :- embed(world, 8).
6 :- embed(is_event, 8).
7 :- event(e, 0).
8 is_event(M,N) :-
      is_process(M), is_type(N)
      :: emb(M,N).
9 e(M,N) :-
      world, is_process(M), is_type(N)
      :: prob(M,N).
10 world <- init.
11 world <- e(M,N), is_event(M,N), world.
```

As §6.1 noted, an event's probability is carried by an `e` fact, but its embedding is carried by an `is_event` fact. This is because the NHP uses dynamic event probabilities (which depend on `world`) but static event embeddings (which do not). Otherwise, we could merge the two by using dimension 8 for `e` in rule 7, and removing `is_event` by deleting it from rule 11 and deleting rules 6 and 8.

F.3. Details of IPTV Dataset and our NDTT Model

For the IPTV domain, the time unit is 1 minute. Thus, in the graph for time prediction, an error of 1.5 (for example) means an error of 1.5 minutes. The exogenous `release` events were not included in the dataset of Xu et al. (2018), but Xu et al. (p.c.) kindly provided them to us.

For our experiments in §6.2, we used the events of days 1–200, days 201–220, and days 221–240 as training, dev and test data respectively—so there is just one long sequence in each case. (We saved the remaining days for future experiments.)

We evaluated the ability of the trained model to extrapolate from days 1–200 to future events. That is, for dev and test, we evaluated the model's predictive power on the held-out dev and test events respectively. However, when predicting each event, the model was still allowed to condition on the *full* history of that event (starting from day 1). This full history was needed to determine the facts in the database, their embeddings, and the event intensities.

Each observed event has one of the forms

```
1 init
2 release(P)
3 watch(U,P)
```

For example, `watch(u4,p49)` occurs whenever user `u4` `watch`es television program `p49`.

The dataset also provides time-invariant facts of the form

```
4 has_tag(P,T)
```

which tag programs with attributes.[26] For example:

```
5 has_tag(p1,comedy).
      ⋮
6 has_tag(p49,romance).
```

We develop our NDTT program as follows. A television program is added to the database only when it is released:

```
7 program(P) <- release(P).
```

Now that `P` is a program, it can be watched:

```
8 watch(U,P) := user(U), program(P).
```

The probability of a watch event depends on the current embeddings of the user and the program:

```
9 embed(user, 8).
10 embed(program, 8).
```

Of course, we have to declare that 'watch' is an event:

```
11 event(watch,8).
```

Notice that we equipped `watch` with a 8-dimensional embedding as well as a probability. The embedding encodes some details of the event (who watched what). This detailed watch event then updates what we know about both the user and the program, in order to predict future watch

[26] Users could also have tags, to record their demographics or interests. However, the IPTV dataset does not provide such tags.

Dataset	$\lvert\mathcal{K}\rvert$	# of Event Tokens			# of Sequences		
		Train	Dev	Test	Train	Dev	Test
Synthetic $M = 4$	16	42000	2100	2100	2000	100	100
Synthetic $M = 8$	32	42000	2100	2100	2000	100	100
Synthetic $M = 16$	64	42000	2100	2100	2000	100	100
IPTV	49000	27355	4409	4838	1	1	1
RoboCup	528	2195	817	780	2	1	1

Table 1. Statistics of each dataset.

events:

```
12 user(U)    <- watch(U,P).
13 program(P) <- watch(U,P).
```

The := connector in rule 8 requested highway connections around watch (Appendix A.3), so these update rules 12 and 13 not only consider ⟦watch(U,P)⟧ but also directly consider ⟦user(U)⟧ and ⟦program(P)⟧. This is similar to a traditional LSTM update, and in our initial pilot experiments we found it to work better than simply using :- in rule 8.

Where do the user facts come from? Rule 12 would automatically add user(U) to the database upon the first time they watched a program. But such an event watch(U,P) is not itself possible (rule 8) until user(U) is already in the database. To break this circularity, we must populate the database with users in advance.

If we simply declared these users as

```
14 user(u1).
15 user(u2).
      ⋮
```

then the model would include separate parameters for each of these rules. However, fitting user-specific parameters would be hard for users who have only a small amount of data. Instead, we make all the user rules share parameters (see Appendix B):

```
16 user(u1) :: user_init.
17 user(u2) :: user_init.
      ⋮
```

Thus, all users start out in the same place,[27] and a users embedding only depends entirely on programs that theyve watched so far. An update to the user's embedding (rule 12) could be either material or epistemic: that is, it may reflect *actual* changes over time in the user's taste, or merely changes in our *knowledge* of the user's taste. Ultimately, the training procedure learns whatever updates help the model to better predict the user's future watch events.

[27] We suspect that it would have been adequate for that initial user embedding to be the **0** vector, which we could have specified by writing :: 0 instead of :: user_init. That is how we treated programs in this model (rule 19 below), and how we treated both users and programs in Appendix F.4. We regret the discrepancy.

There is one more subtlety regarding user embeddings. In the program above, user(u1) is true at all times, but is "launched" (in the sense of §3.3) only by the first event of the form watch(u1,P). Thus, we learn nothing about the user from the fact that time has elapsed without their having yet watched any programs: they do not yet have a cell block that can drift to track the passage of time. To fix this, we add the following rule so that all users are simultaneously launched at time 0 by the exogenous init event:

```
18 user(U) <- init, user(U).
```

This ensures that the user has an LSTM cell block starting at time 0, which can drift to mark the passage of time even before the user has watched any programs. This rule for users is analogous to rule 7 for programs.

Where do the program facts come from? We declare them much as we declared the user facts:[28]

```
19 program(p1) :: 0.
20 program(p2) :: 0.
      ⋮
```

However, a program's embedding should also be affected by its tags:[29]

```
21 program(P) :- has_tag(P,T), tag(T).
```

where each tag is declared separately:

```
22 embed(tag, 8).
23 tag(adventure).
24 tag(comedy).
      ⋮
```

Note that the rules like 23 and 24 introduce tag-specific parameters. For example, the bias vector of rule 23 provides an embedding of the adventure tag. As each tag has a lot of data, these tag-specific parameters should be easier to learn than user-specific parameters.

The initial embedding of a tag is then affected by who watches programs with that tag, and when. In other words,

[28] Actually, if p1 has at least one tag, then we can omit rule 19 because rule 21 below will be enough to prove that p1 is a program. In the IPTV dataset, every program does have at least one tag, so we omit all rules like 19, which do not affect the facts or their embeddings.

[29] Recall that facts like has_tag(p1,comedy) were declared in the initial database, have no embeddings, and never change.

just as the watch events update our understanding of individual users, they also track how the meaning of each tag changes over time:

```
25 tag(T) <- init, tag(T).
26 tag(T) <- watch(U,P), has_tag(P,T), tag(T).
```

As before, these updates are rich because the watch event has an embedding and also supplies highway connections.

We finish with a final improvement to the model. Above, program(P) is affected both by P's tags via the :- rule 21 and by its history of watch events via the <- rule 13. The NDTT equations would simply add these influences via rule (7). Instead, we edit the program to combine these influences nonlinearly. This gives a deeper architecture:

```
27 program(P) :-
       program_profile(P), program_history(P).
28 program_profile(P) :- has_tag(P,T), tag(T).
29 program_history(P) <- release(P).
30 program_history(P) <-
       watch(U,P), user(U), program(P).
```

where rules 28–30 replace rules 21, 7, and 13 respectively.

In principle, facts with different functors can be embedded in vector spaces of different dimensionality, as needed. But in all of our experiments, we used the same dimensionality for all functors, so as to have only a single hyperparameter to tune. If the hyperparameter were 8, for example, our Datalog program would have the declarations

```
31 :- embed(user, 8).
32 :- embed(program, 8).
33 :- embed(profile, 8).
34 :- embed(released, 0).
35 :- embed(watchhistory, 8).
36 :- embed(tag, 8).
37 :- event(watch, 8).
```

where watch has an extra dimension for its intensity. The hyperparameter tuning method and its results are described in Appendix F.7 below.

F.4. Baseline Programs on IPTV Dataset

We also implemented baseline models that were inspired by the Know-Evolve (Trivedi et al., 2017) and DyRep (Trivedi et al., 2019) frameworks. Our architectures are not identical: for example, our rule 3 below models each event probability using a feed-forward network in place of a bilinear function. However, Trivedi (p.c.) agrees that the architectures are similar. Note that these prior papers did not apply their frameworks specifically to the IPTV dataset (nor to RoboCup).

The Know-Evolve and DyRep programs specify the same user, program, and has_tag facts as in Appendix F.3, except that the initial embedding user_init is fixed to **0** (see footnote 27).

The Know-Evolve program continues as follows.

Whereas a watch fact in Appendix F.3 carried both a probability and an embedding, here we split off the embedding into a separate fact and compute it differently from the probability, to be more similar to Trivedi et al. (2017):

```
1 :- event(watch, 0).
2 :- embed(watch_emb, 8).
3 watch(U,P) :- user(U), program(P).
4 watch_emb(U,P) :-
      user(U) : pair, program(P) : pair.
```

Notice that rule 4 in effect multiplies the sum $[\![\texttt{user(U)}]\!] + [\![\texttt{program(P)}]\!]$ by the pair matrix before applying $\tanh$.

The cell blocks are now launched and updated as follows:

```
5 user(U)    <- init, user(U).
6 program(P) <- init, program(P).
7 user(U)    <- watch(U,P), watch_emb(U,P).
8 program(P) <- watch(U,P), watch_emb(U,P).
```

Of course, when the embedding of user(U) or program(P) is updated, the embedding of watch_emb(U,P) also changes to reflect this.

What are the differences from Appendix F.3? Since Trivedi et al. (2017) did not support changes over time to the set of possible events, we omitted this feature from our Know-Evolve program above. Specifically, the program does not use the release events in the dataset—it treats all programs as having been released by init at time 0. The program also has no highway connections, nor the deeper architecture at rules 27–30 of Appendix F.3, and it does not make use of the program tags.

Our DyRep version of the program makes a few changes to follow the principles of (Trivedi et al., 2019). The main ideas of DyRep are as follows:

- Entities are represented as nodes in a graph (here: programs, users, and tags).
- Each node has an embedding.
- The properties of an entity are represented by labeled edges that link it to other nodes (here: has_tag(P,T)).
- The graph structure can change due to exogenous forces (see rule 9 below).
- Any pair of entities can communicate at any time. (These communications are the events in our temporal event sequences, such as watch(U,P).)
- The probability of an event depends on the embeddings of the two nodes that communicate (here: rule 3).
- When an event occurs, it updates the embeddings of (only) the two nodes that communicate (see rules 10 and 11 below).
- An update to a node's embedding also considers the embeddings of its neighbors in the graph (see rule 12 below).

Thus, we replace rules 6–8 above with

```
program(P) <- release(P).
user(U)    <- watch(U,P), user(U)    :: event.
program(P) <- watch(U,P), program(P) :: event.
```

Thus, DyRep now permits the set of watchable programs (nodes) to change over time, but the `user` and `program` updates are less well-informed than in Know-Evolve: the updates to the user embedding no longer look at the current program embedding, nor vice-versa.[30] Indeed, DyRep no longer uses `watch_emb` and can drop rule 4.

Where our Know-Evolve program did not use tags, our DyRep program can encode tags using `has_tag` edges. Thus, when a program `P` is watched, the update to the program's embedding depends in part on its tags:

```
program(P) <-
    watch(U,P), tag(T), has_tag(P,T).
```

The embedding $[\![\texttt{tag(T)}]\!]$ is defined as in our full model of Appendix F.3, except that it is now static (except for drift). It is no longer updated by watch events, because the `watch(U,P)` event only updates `U` and `P`. In contrast, the Datalog rule 26 in Appendix F.3 was able to draw `T` into the computation by joining `watch(U,P)` to `has_tag(P,T)`.

F.5. Details of RoboCup Dataset and our NDTT Model

For the RoboCup domain, the time unit is 1 second. Thus thus in the graph for time prediction, an error of 1.5 (for example) means an error of 1.5 seconds.

For our experiments in §6.2, we used Final 2001 and 2002, Final 2003, and Final 2004 as training, dev, and test data respectively. Each sequence is a single game and each dataset contains multiple sequences.

Each observed event has one of the forms

```
kickoff(P)
kick(P)
goal(P)
pass(P,Q)
steal(Q,P)
init
```

which we will describe shortly. The database also contains facts about the teams. There are 2 teams, each with 11 robot players. Any pair of players `P` and `Q` are either teammates or opponents:

```
teammate(P,Q) :-
    in_team(P,T), in_team(Q,T), not_eq(P,Q).
opponent(P,Q) :-
    in_team(P,T), in_team(Q,S), not_eq(T,S).
```

[30] To allow better-informed updates within the DyRep formalism, we could have included edges between all users and all programs. But then every update would depend on all users and all programs—which is exactly the "everything-affects-everything" problem that our paper aims to cure (§1)!

These relations are induced using the database facts

```
in_team(a1,a).
    ⋮
in_team(a11,a).
in_team(b1,b).
    ⋮
in_team(b11,b).
```

together with an inequality relation on entities, `not_eq`, which can be spelled out with a quadratic number of additional facts if the Datalog implementation does not already provide it as a built-in relation:

```
not_eq(a1, a2).       % players
not_eq(a1, a3).
    ⋮
not_eq(b11, b10).
not_eq(a, b).         % teams
not_eq(b, a).
```

We allow the ball to be in the possession of either a specific player, or a team as a whole. A game starts with team `a` taking possession of the ball:[31]

```
has_ball(a) <- init.
```

A random player `P` in team `a` now assumes possession of the ball, taking it from the team as a whole.[32] This is called a `kickoff` event, although in RoboCup—unlike human soccer—`P` does not kick the ball off into the distance but retains it.

```
kickoff(P) :- in_team(P,T), has_ball(T).
!has_ball(T) <- kickoff(P), in_team(P,T).
has_ball(P)  <- kickoff(P).
```

Thereafter, the player who has possession of the ball can kick it to a nearby location while retaining possession ("dribbling"),

```
kick(P) :- has_ball(P).
```

or can pass the ball to a teammate,

```
pass(P,Q) :- has_ball(P), teammate(P,Q).
!has_ball(P) <- pass(P,Q).
has_ball(Q)  <- pass(P,Q).
```

or can score a goal,

```
goal(P) :- has_ball(P).
```

Scoring a goal instantly updates the database to transfer the ball to the other team,

[31] It is a convention in the IPTV dataset that team `a` is the one that takes possession first. If the starting team were decided by a coin flip, then we would use the "event groups" extension in Appendix A.6 to decide whether `init` causes `has_ball(a)` or `has_ball(b)`. This would allow us to learn the weight of the coin (for example, on the IPTV dataset, we would learn that the coin *always* chooses team `a`); or if we knew it was a fair coin, we could model that by declaring that certain parameters are 0.

[32] Notice that in our program, the possible kickoff events all have equal intensity, leading to a uniform distribution over players `a1`, ..., `a11`. We will learn that this intensity is high, since the kickoff happens at a time close to 0.

```
!has_ball(P) <- goal(P).
has_ball(S)  <- goal(P), in_team(P,T),
      not_eq(T,S).
```

after which someone in the other team can kick off the ball and continue the game. When a player P has the ball, a player Q in the other team can steal it:

```
steal(Q,P) :- has_ball(P), opponent(P,Q).
!has_ball(P) <- steal(Q,P).
has_ball(Q)  <- steal(Q,P).
```

In our experiments, we got the best results by declaring non-zero embeddings of both teams and players, such as

```
:- embed(team, 8).
:- embed(player, 8).
```

Since there are only two teams, the embeddings of the two teams jointly serve as a kind of global state—but one that may be smaller than the global state we would use for a simple NHP model. In our actual experiments (§6.2), hyperparameter search (Appendix F.7) chose 32-dimensional NDTT embeddings, giving a total of 64 dimensions for the pair of teams. In contrast, it chose a 128-dimensional global state for the simple NHP baseline model.

Ideally, we would like the embedding $[\![\texttt{player(P)}]\!]$ to track our probability distribution over the state of the robot player, such as its latent position on the field and its latent energy level. We would also like the embedding of a team to track our probability distribution over the state of the team and the latent position of the ball. We do not observe these latent properties in our dataset. However, they certainly affect the progress of the game. For example, if two players pass or steal, they must be near each other; so if we have pass(P,Q) and steal(R,Q) nearby in time, then by the triangle inequality, P and R must be close together, which raises the probability of steal(P,R). Changes in the mean and variance of these probability distributions are then tracked by updates and drift of the embeddings, with the variance generally decreasing when an event occurs (because it gives information) and increasing between events (because uncertainty about the latent changes accumulates over time, as in a drunkards walk).

The team and player embeddings are launched at time 0 using the exogenous init event:

```
team(T)   <- init, in_team(P,T).
player(P) <- init, in_team(P,T).
```

A players embedding is updated whenever that player participates in an event. We elected to reduce the number of parameters by sharing parameters not only across players, but also across similar kinds of events (this was also done by the prior work DyRep).

```
player(P) <- kickoff(P) :: individual.
player(P) <- kick(P)    :: individual.
player(P) <- goal(P)    :: individual.
player(P) <- pass(P,Q)  :: individual_agent.
player(Q) <- pass(P,Q)  :: individual_patient.
player(Q) <- steal(Q,P) :: individual_agent.
player(P) <- steal(Q,P) :: individual_patient.
```

The parameter sharing notation was explained in Appendix B. The above rules use the linguistic names "agent" and "patient" to refer to the player who acts and the player who is acted upon, respectively.

A teams embedding is also updated when any player acts. We could have done this by saying that the teams embedding pools over all of its players, so it is updated when they are updated,

```
team(T) :- player(P), in_team(P,T)
```

but instead we directly updated the team embeddings using update rules parallel to the ones above. For example, rule 37 also has a variant that affects not the player P that kicked the ball, but that player's team T, as well as a second variant that affects the opposing team.

```
team(T) <-
      kick(P), in_team(P,T) :: team.
team(S) <-
      kick(P), in_team(P,T), not_eq(T,S)
      :: team_other.
```

We similarly have variants of rules 39–42:

```
team(T) <-
      pass(P,Q), in_team(P,T)
      :: team_agent.
team(S) <-
      pass(P,Q), in_team(P,T), not_eq(T,S)
      :: team_nonagent.
team(T) <-
      steal(P,Q), in_team(P,T)
      :: team_agent.
team(S) <-
      steal(P,Q), in_team(P,T), not_eq(T,S)
      :: team_nonagent.
```

Here "non-agent" refers to the team that does not contain the agent (in the case of rule 47, it does not contain the patient either).

Finally, we can improve the model by enriching the dependencies. Earlier, we embedded the kick event using rule 22, repeated here:

```
kick(P) :- has_ball(P).
```

But then the probability that robot player P kicks at time t (if it has the ball) would be constant with respect to both P and t. We want to make this probability sensitive to the states at time t of the player P, the players team T, and the other team S. So we modify the rule to add those facts as conditions (in blue):

```
kick(P) :=
      has_ball(P), player(P), team(T), team(S),
      in_team(P,T), not_eq(T,S).
```

Because this rule uses := to request highway connections, all three of these states will also be consulted directly when a kick(P) event updates the states of player P and both

teams (via rules 22, 44 and 45). To deepen the network, we further give the event kick(P) its own embedding, which is a nonlinear combination of all of these states, and which is also consulted when the event causes an update.

```
52 | :- event(kick,8).
```

We handle the other event types similarly to kick. In the case of an event that involves two players P and Q, we also add the state of player Q (the patient) as a fourth blue condition. For example, we expand the old rule 23 to

```
53 | pass(P,Q) :-
       has_ball(P), teammate(P,Q), player(P),
       player(Q), team(T), team(S), has_ball(P),
       in_team(P,T), not_eq(T,S).
```

F.6. Baseline Programs on RoboCup Dataset

As before, we also implemented baseline models that are inspired by the Know-Evolve and DyRep frameworks (Trivedi, p.c.). The non-embedded database facts about players and teams are specified just as in Appendix F.5 (rules 7–17).

Like the Know-Evolve program for IPTV, the Know-Evolve program for RoboCup has no embeddings for its events:

```
1 | :- event(kickoff, 0).
2 | :- event(kick, 0).
3 | :- event(goal, 0).
4 | :- event(pass, 0).
5 | :- event(steal, 0).
```

As in IPTV, the embeddings are handled by separate facts. Know-Evolve's embedding of an event does not depend on the event's type, but only on its set of participants. Thus, the kickoff, kick, and goal events are simply represented by the embedding of the single player that participates in those events, which is defined exactly as in our full model of Appendix F.5:

```
6 | :- embed(player, 8).
7 | player(P) <- init, in_team(P,T).
```

For the pass and steal events, we also need an embedding for each *unordered pair* of players (analogous to watch_emb in Appendix F.4 rule 4):

```
8 | :- embed(players, 8).
9 | players(P,Q) :-
      player(P) : pair, player(Q) : pair.
```

All of these embeddings evolve over time. Since teams do not participate directly in events, they do not have embeddings, in contrast to our full model in Appendix F.5.

Each event's probability depends nonlinearly on the concatenated embeddings of its participants, e.g.,

```
10 | kick(P)    :- player(P).
11 | pass(P,Q) :-
       player(P), player(Q), teammate(P,Q).
```

Note that because Know-Evolve does not allow changes over time in the set of possible events, it assigns a positive probability to the above events even at times when P does not have the ball.

Actually, Trivedi et al. (2017, 2019) allow any event to take place at any time between any pair of entities. Our Know-Evolve and DyRep programs take the liberty of going beyond this to impose some *static* domain-specific restrictions on which events are possible. For example, in RoboCup, rule 11 only allows passing between teammates, and rule 10 only allows kicking from a player to itself (i.e., the "pair" of participants for kick(P) has only one unique participant).

An event updates the embeddings of its participants, e.g.,

```
12 | player(P) <- : kick
       kick(P), player(P) : only.
13 | player(P) <- : pass
       pass(P,Q), players(P,Q) : agent.
14 | player(Q) <- : pass
       pass(P,Q), players(P,Q) : patient.
```

where the bias vector is determined by the event type (e.g., kick or pass), while the weight matrix is determined by the role played in the event of the participant being updated (agent, patient, or only—see Appendix F.5). Both types of parameters are shared across multiple rules.

For the DyRep program, the same events are possible as for Know-Evolve, and most of the rules are the same. However, recall from Appendix F.4 that DyRep permits us to define a graph of entities. Robot players are entities, of course. We also consider the ball to be an entity, which is connected to player P by an edge when P possesses the ball. This allows DyRep to update the embeddings of the participants in a pass or steal event to record the fact that the one who had the ball now lacks it, and vice-versa. The model can therefore learn that pass(P,Q) and steal(Q,P) are much more probable when P has the ball.

DyRep requires the following new rules to handle the ball:

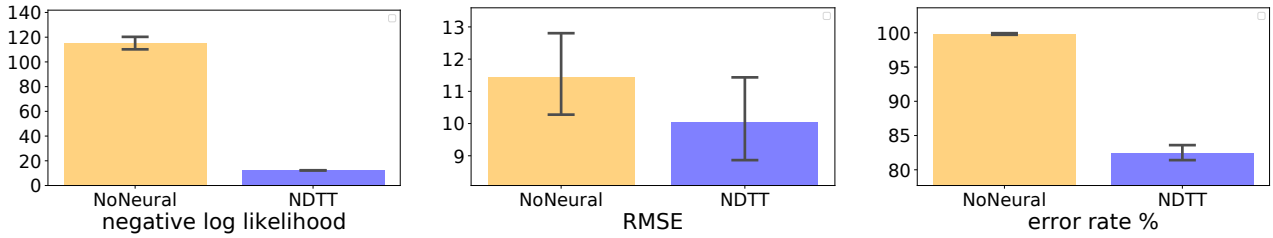
```
15 | :- embed(ball, 8).
16 | ball <- init.
```

as well as all of the rules from Appendix F.5 that update has_ball, which manage the edges of the evolving graph. Note that ⟦ball⟧ may drift over time but is never updated, since ball is never one of the participants in an event.

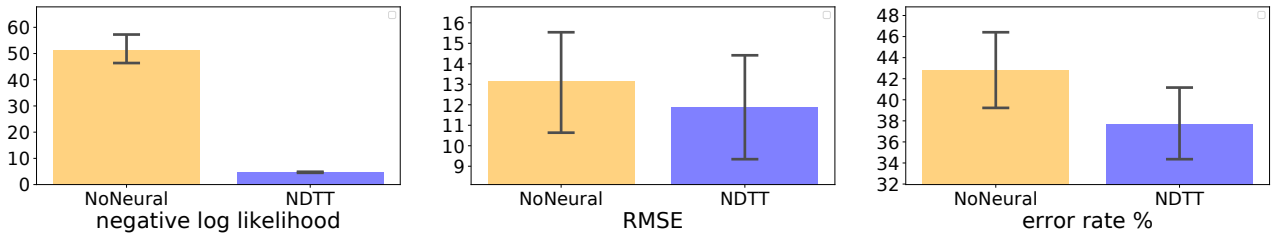
Now we mechanically obtain the DyRep model by replacing Know-Evolve rules such as rules 12–14 with DyRep-style versions:

```
17 | player(P) <- : kick
       kick(P),   player(P) :: event.
18 | player(P) <- : pass
       pass(P,Q), player(P) :: event.
19 | player(Q) <- : pass
       pass(P,Q), player(Q) :: event.
```

and then mechanically adding influences from the neighbors of P and Q (where the ball is the only possible neigh-



(a) IPTV Dataset



(b) RoboCup Dataset

Figure 4. Ablation study of taking away neural networks from our Datalog programs in the real-world domains. The format of the graphs is the same as in Figure 2. The results imply that neural networks have been learning useful representations that are not explicitly specified in the Datalog programs.

bor):

```
20 player(P) <-
       kick(P), ball : ball, has_ball(P).
21 player(P) <-
       pass(P,Q), ball : ball, has_ball(P).
22 player(Q) <-
       pass(P,Q), ball : ball, has_ball(Q).
```

Remarks. Recall that the DyRep model can unfortunately generate domain-impossible event sequences in which `P` kicks or passes the ball without actually having it. However, such events never happen in *observed* data. As a result, the above rules can be simplified if we are only updating embeddings based on observed events (which is true in our experiments). We can then remove the explicit `has_ball(P)` condition from rules 20 and 21 because it is surely true when these rules are triggered by observed events. And we can remove rule 22 altogether, because its condition `has_ball(Q)` is surely false when this rule is triggered by an observed event. But then `has_ball` plays no role in the DyRep model anymore! This shows that in effect, the model tracks the ball's possessor only by updating `player(P)` whenever it observes an event with participant `P` in which `P` has the ball. This type of tracking is imprecise (in particular, it does not immediately detect when `P` *acquires* the ball), which is why the DyRep model cannot learn from data to assign probability ≈ 0 to domain-impossible events.

F.7. Training Details

For every model in §6, including the baseline models, we had to choose the dimension D that is specified in the `embed` and `event` declarations of its NDTT program. For simplicity, all declarations within a given program used the same dimension D, so that each program had a single hyperparameter to tune. We tuned this hyperparameter separately for each combination of program, domain, and training size (e.g., each point in Figure 1 and each bar in Figures 2, 3 and 4), always choosing the D that achieved the best performance on the dev set. Our search space was $\{4, 8, 16, 32, 64, 128\}$. In practice, the optimal D for a model of a non-synthetic dataset (§6.2) was usually 32 or 64.

To train the parameters for a given D, we used the Adam algorithm (Kingma & Ba, 2015) with its default settings and set the minibatch size to 1. We performed early stopping based on log-likelihood on the held-out dev set.

F.8. Ablation Study II Details

In the final experiment of §6.2, all embeddings have dimension 0. Each event type still has an extra dimension for its intensity (see §3.2). The set of possible events at any time is unchanged. However, the intensity of each possible event now depends only on *which* rules proved or updated that possible event (through the bias terms of those rules); it no longer depends on the embeddings of the specific atoms on the right-hand-sides of those rules. Two events may nonetheless have different intensities if they were proved by different `:-` rules, or proved or updated by different sequences of `<-` rules (where the difference may be in the identity of the `<-` rules or in their timing).

Our experimental results in Figure 4 show that the neural networks have really been learning representations that are actually helpful for probabilistic modeling and prediction.